Database Standards

Version 1.0

Modified: 4/23/2015

Table of Contents

1. Database

1.1 Creating and Naming of Databases

Standards:

1.1.1 Items to consider before creating a new database:

a) The scope of the project needs to be understood and the key players identified.
b) Limit the number of databases. Based on the project scope, see if there is an existing database available.
c) The size of the project. If the project only requires a few tables, 20 or less, locate another database that closely matches the scope to store the project data instead of creating a new database.
d) Confidentiality needs may override some of the above rules. Contact DB Administrator for guidance.

1.1.2 Choosing a Database Name

a) While focusing on the items from Section 1.1.1, the database name should reflect NDE's Program Level. If the scope involves multiple Program Groups, the database name should reflect a Project Level perspective.
b) Limit database name to a maximum of 25 characters long.
c) Acronyms should be in all capital letters. Reference the "List of NDE Acronyms Appendix" for acronyms currently in use.
d) Capitalize the first letter of each word in the database name, but lowercase the rest of the letters in the word.
e) Do not use spaces in the name of a database.
f) Do not use underscores (_) to indicate spaces between words.
g) Do not use special characters in the name of the database.

1.2 Database File Name

Standards:

a) The data and transaction log files must match the database name.

1.3 Database File Locations – Database Server(s) Only

Standards:

a) Transaction logs are located on E drive in the following directory structure: E:\MSSQL\Log
b) Data files are located on the F drive in the following directory Structure: F:\MSSQL\Data

2. Tables and Views

2.1 Table and View Naming

Best Practices:

a) Normalization of tables should be used for larger applications and data collections based on scope and size (number of rows and columns).
    i. Examples of larger systems include:
        1. Teacher Certification System, Adult Education, Nutrition Services System, Grants Management System
b) De-normalization of tables should be used when summarizing and reporting of Information/data. When trying to decide to de-normalize, focus on the audience requirements.
    i. Examples of Warehouse systems:
        1. DataWarehouse and NDE_MART
c) If a global temp table is needed, use a drop table at the end of your process. Please contact the Database Administrator for guidance.
d) It is highly recommended to limit the table and view name length to 50 characters or less.
e) Views can be used to hide certain columns in a table.
f) Views can be used to give different perspectives of the same table
g) Views can be used to group frequently accessed tables that are joined together to make query writing sampler. (Minding performance issues)

Standards:

a) Keep the table naming consistent throughout the database.
    i. Following the naming convention/format within the database that you are working in.
        a. Example: prefix, suffix, case, acronyms, etc…
b) If the table is for a temporary purpose, like an import from an Excel file or external data source, for a single use, start the naming of the table with the underscore (_). This is to identify tables to be dropped later.
        a. Example: _FreeLunchCounts
c) Do not include datayears in table names, but some exceptions could include single purpose data.
d) Table name should mirror its content, function, and meaning.
e) Use the default ‘dbo’ schema. Do not create additional schemas.
f) Acronyms should be in all capital letters. Reference the “List of NDE Acronyms Appendix” for acronyms currently in use.
g) Capitalize the first letters of each word in the table name, but lowercase the rest of the letters in the word.
h) Keep to letters and numbers

i) Underscores (_) are acceptable to clarify as needed, and use underscores after a prefix. Do **not** use underscores to separate words, and do not use spaces and any other special characters.

EXAMPLE: ADA_Students8_12 versus ADA_Students812

j) Use a prefix to group tables together based on project. Use the project name, the data source, or an NDE Acceptable Acronym. Reference the "List of NDE Acronyms Appendix" for acronyms currently in use.
k) When using global temp tables, use unique names so that multiple process do not interfere with each other.
l) When creating new tables, do **not** prefix or suffix the table name with the letters 'TBL'. The only exception to this rule is when you are importing tables from an outside source, which you can retain the name of the data source
m) Do not use views to update tables
n) View names follow the table naming conventions, but they MUST be prefixed with a lower case '**vw_**'.

EXAMPLE: dbo.vw_ECPRS_EducationInstitutions

3. Columns, Data Types, and Keys

3.1 Column Naming

Best Practices:

a) It is highly recommended to limit the column name length to 20 characters or less.
b) It is not recommended to use a data type prefix to name your columns.
c) Prefer to not include DATAYEARS (example 20102011) in column names.
d) When using an Identity field, try to relate to the table name when possible or a generic version if needed should be just ID

1. "Tablename"ID EXAMPLE: UserTypeID

e) Table logging fields are recommended, but not required. LastUpdated, LastUpdatedBy, CreatedDate, CreatedBy, or Procedure_Name. Helpful for dataflow tracking and timing.
f) Generally used Reference columns:

Datayears = CHAR (8) '20112012'
AgencyID = Char (11) '55-0001-000'
County = Char(2)
CoDIST = CHAR(6 or 7)
District = CHAR(4)
DistrictCode = Char(7) 'xx-xxxx'
School = CHAR(3)
Agrmbr = CHAR(6)

FiscalYrNbr = Numeric
School_Year = DateTime
NDE_STUDENT_ID = VARCHAR (10)
STUDENT_ID = CHAR or VARCHAR(10 or 12) – NSSRS
NDE_STAFF_ID = CHAR(10)

Standards:

a) Keep the column naming consistent throughout the tables
    i. Following the naming convention/format within the database that you are working in.
b) Do not use Reserved or Forbidden words for column names. If you have to bracket the field, use a different name. Reference "Reserved or Forbidden Words Appendix" for a complete listing:

    Examples of what not to use: Procedure, Date, Case

c) Acronyms should be in all capital letters. Reference the "List of NDE Acronyms Appendix" for acronyms currently in use.
d) Capitalize the first letters of each word in the table name, but lowercase the rest of the letters in the word.
e) Keep to letters and numbers
f) Do not start a column name with a number.
g) Underscores (_) are acceptable to clarify as needed, but do **not** use underscores to separate words. Do not use spaces and any other special characters.

3.2 Data Types

Best Practices:

a) Nulls may be used in a variety of way
    Date/Time situations
    To represent absence of data
    When a value is not applicable

Standards:

a) Number must be "a" numeric data type that can be used in calculations

    Floats, reals can cause issues with rounding
    Decimal rounding issues and truncations
    The precision will be depended upon program needs.

b) String characters use VARCHAR to save space when using non fixed length data values.

c) CHAR is only used for fixed length strings that never have a blank value.
Example: Student_ID, NDE_Staff_ID
d) Use Datetime for dates and or time values. Do not store as a string
e) Yes / No are stored as Char (1). Example: Y/N
f) Tri state would be stored as VARCHAR(1). Example: Y/N/"Blank"

3.3 Keys

Standards:

3.3.1 Primary Keys - Unique Values for each record; could be composite key or identity field.

a) Keep the current database's naming convention. For new, do the following:
b) Prefix Primary Keys with PK_.
c) After the underscore, name the rest of the Primary Key after the table name. (Follow the case and characters of the table name.)
d) Primary Keys are required.

3.3.2 Foreign Keys - a field (or collection of fields) in one table (Child Table) that uniquely identifies a row of another table (Parent Table)

a) Keep the current database's naming convention. For new, do the following:
b) Prefix with FK_
c) After the underscore, name it using the Parent Table name. (Follow the case and characters of the table name.)
d) With multiple foreign keys, keep the name unique.
e) Foreign Keys are NOT required.

4. Indexes

Best Practices:

a) Indexes can be applied to larger temp tables to improve performance. If the table is a global temp table, make sure to drop it at the end of your process.

Standards:

a) Prefix the indexes using the following matrix. Uppercase the prefix:

UDX_ – Unique Clustered index
IDX_ – Non Unique Clustered Index
UNX_ – Unique Non Clustered Index
INX_ – Non Unique Non Clustered Index

b) Follow by the column name or the abbreviated versions of the column names separated by underscores

c) Order the names by how they are indexed. This will accommodate multiple indexes using the same columns in different order

Example: IDX_LastName_FirstName
IDX_FirstName_LastName

5. Stored Procedures

5.1 User Defined Stored Procedures (USP) Naming

Best Practices:

a) Try to name stored procedures Noun/Verb for readability.
b) In SQL Server, the sp_ prefix designates system stored procedures. If you use that prefix for your stored procedures, the name of your procedure might conflict with the name of a system stored procedure that will be created in the future.

Standards:

a) Keep the current database's naming convention. For new, do the following:
b) Do not use cursors or while loops due to performance issues unless they are unavoidable. Please contact the DBA.
c) Prefix stored procedure *name* with lowercase usp_
d) Use the default 'dbo' schema. Do not create additional schemas.
e) Use a prefix to group procedures together based on project. Use the project name, the data source, or an NDE Acceptable Acronym. Reference the "List of NDE Acronyms Appendix" for acronyms currently in use.
f) Name the stored procedure based on activity (i.e. usp_FTE_Calculate)
g) Acronyms should be in all capital letters. Reference the "List of NDE Acronyms Appendix" for acronyms currently in use.
h) Capitalize the first letters of each word in the stored procedure name, but lowercase the rest of the letters in the word.
i) Keep to letters and numbers
j) Do not start a name with a number.
k) Do not drop or create permanent/physical tables from within procedures.

5.2 Formatting Stored Procedures

Standards:

a) Add a comments section at the top with the following information:

Author
Creation date
Definition / Purpose
Change history –Date, First and Last Name,- and description of the change

EXAMPLE:

```
-- ============================================
-- Author:  John Doe
-- Create date: 6/17/2013
-- Description:    Return Cohort details for single group for multiple years
--2010-04-12 John Doe - (A Added a statement to update nde_student_id with the student_id_active
--                      from student_id_retired_active table for longitudinal analysis
--2010-08-27 Sally Smith -  Added code to incorporate Nesa reading data into student_summary
--2010-08-30 John Doe -  Changed table structure
                        -- Added NESA_FULL_ACADEMIC_YEAR_DISTRICT
                        -- Added NESA_FULL_ACADEMIC_YEAR_SCHOOL
--                      Also Added code to update the columns with NESA Reading data from
                        --NESA_STUDENT_INDICATORS.
-- ============================================
```

b) Comment throughout the code to explain what the code is doing.

c) SQL code should be formatted for ease of readablity, and formatting tools are avaible on the Internet if needed.

d) Tips on optimizing for performance, reference the Code Optimization section.

5.3 Stored Procedures to move data

(Reference Extract, Transform, and Load (ETL) section)

6. Triggers

6.1 When to use Triggers

Best Practices:

a) Avoid using triggers with deletes. Contact the DBA
b) It is highly recommended to use triggers only if you really need, and you should opt to use stored procedures instead if possible.
c) They are great for date change items and who made the changes for logging or history purposes.
d) Tips on optimizing for performance, reference the Code Optimization section.

6.2 Trigger Naming

Standards:

a) Keep the current database's naming convention. For new, do the following:
b) Do not use cursors or while loops due to performance issues unless they are unavoidable. Please contact the DBA.
c) Prefix it with a lower case tr_
d) Use the default 'dbo' schema. Do not create additional schemas.
e) Followed by 'column name'
f) Follow the case and characters of the column name.

Example: tr_LastUpdatedBy

7. Functions

7.1 User Defined Function (UDF) Naming

Standards:

a) Keep the current database's naming convention. For new, do the following
b) Do not use cursors or while loops due to performance issues unless they are unavoidable. Please contact the DBA.
c) Follow most current ANSI standard verses using specialty system functions. Built-in Functions may change or be removed by the manufacturer (IBM, MSSQL, Oracle) in the new versions of the Database System software.
d) Prefix function *name* with lowercase fn_
e) Use the default 'dbo' schema. Do not create additional schemas.
f) Use a prefix to group functions together based on project. Use the project name, the data source, or an NDE Acceptable Acronym. Reference the "List of NDE Acronyms Appendix" for acronyms currently in use.

g) Name non-project specific functions based on activity (i.e. fn_AllCaps) Store them in the NDECommon database on NDESQL01 server. Functions in this database will not reference items outside of this database.

h) Capitalize the first letters of each word in the function name, but lowercase the rest of the letters in the word.

i) Keep to letters and numbers

j) Do not start a name with a number.

7.2 Formatting User Defined Functions

Standards:

a) Add a comments section at the top with the following information:

Author
Creation date
Definition / Purpose
Change history –Date, First and Last Name,- and description of the change

EXAMPLE:

```
-- ============================================
-- Author:  John Doe
-- Create date: 6/17/2013
-- Description:   Return Cohort details for single group for multiple years
--2010-04-12 John Doe - (A Added a statement to update nde_student_id with the student_id_active
--                     from student_id_retired_active table for longitudinal analysis
--2010-08-27 Sally Smith -  Added code to incorporate Nesa reading data into student_summary
--2010-08-30 John Doe -  Changed table structure
                       -- Added NESA_FULL_ACADEMIC_YEAR_DISTRICT
                       -- Added NESA_FULL_ACADEMIC_YEAR_SCHOOL
 --                    Also Added code to update the columns with NESA Reading data from
                       --NESA_STUDENT_INDICATORS.
-- ============================================
```

b) Comment throughout the code to explain what the code is doing.

c) SQL code should be formatted for ease of readablity, and formatting tools are available on the Internet if needed.

d) Tips on optimizing for performance, reference the Code Optimization section.

8: SQL Optimization

8.1 Using Local @Table Variables and #Temp Tables

Best Practices:

Local tables created as variables (with @s) cannot use any of the table optimizations built into SQL Server like indexing, unlike temp tables (with #s), and so are only recommended for very small amounts of data.

8.1.1 Explicit Creation vs. Into Clause

a) It can be convenient to create a #temp table using a *SELECT… INTO* statement, but this is only recommended when the #temp table will be relatively simple and all of the columns have defined types. In this example the data types are defined by the source table:

```
select lu.Code,
        lu.Literal
into #luPositionCodeByYear
from DataWarehouse.dbo.luPositionCodeByYear lu with(nolock)
where DataYears='20142015'

...
drop table #luPositionCodeByYear
```

b) In the example below, the *ColumnName* field has no explicitly defined type and so the system creates a default char type for it. This could result in errors if other rows are added to this table later.

```
select 'Position_Code' as ColumnName,
        lu.Code,
        lu.Literal
into #luPositionCodeByYear
from DataWarehouse.dbo.luPositionCodeByYear lu with(nolock)
where DataYears='20142015'

...
drop table #luPositionCodeByYear
```

c) Often, it is preferred to explicitly create the columns and data types for a #temp table before inserting rows into it. The data types on the columns should exactly match the type of the source column, for example if the source table uses VARCHAR(50) then your #temp table shouldn't be VARCHAR (20). This will help avoid unexpected errors with data types and is also much easier for a developer to maintain.

```
create table #luPositionCodeByYear (
        ColumnName varchar(30),
        Code char(4),
        Literal varchar(50) )

insert into #luPositionCodeByYear
...
drop table #luPositionCodeByYear
```

8.1.2 #temp Table Indexing

a) #Temp tables are more efficient than @table variables by default, but you can optimize #temp tables even further by creating a basic index on it, or a fully defined constraint. This would only be recommended for very large/complex tables though. Here is an example of a quick index:

```
create table #luPositionCodeByYear (
        ColumnName varchar(30),
        Code char(4),
        Literal varchar(50) )

create index idx_luPositionCodeByYear on #luPositionCodeByYear (ColumnName, Code)
...
```

8.1.3 Linked Server Queries

a) A key use for #temp tables is for queries across linked servers. Pulling data across servers is highly inefficient and the server does not automatically optimize these queries. The result is that if you use a linked server within a query (such as in a JOIN) the system may have to make thousands of individual trips across servers, and a query that would otherwise takes less than a second could take minutes:

```
select spi.NDE_Staff_ID,
        sg.COURSE_KEY
from DataCenter.dbo.vwStaff_Personal_Information spi with(nolock)
inner join ndesql04.STG_MART.SCHOLWHS.STAFF_SNAPSHOT ss with(nolock)
        on ss.STAFF_ID = spi.NDE_Staff_ID
        and right(spi.DataYears,4) = cast(year(ss.SCHOOL_YEAR) as varchar)
inner join ndesql04.STG_MART.SCHOLWHS.STUD_GRADES sg with(nolock)
        on sg.STAFF_KEY = ss.STAFF_KEY
         and sg.SCHOOL_YEAR = ss.SCHOOL_YEAR
where spi.DataYears = @Datayears
```

b) Instead, as much of the data that comes from the linked server as possible should first be put into the #temp table while using the least amount of cross-server joins. Also, only select the columns for your temp table that are needed. Then you can use the #temp table as a direct replacement in the original query:

```
select ss.STAFF_ID,
        sg.COURSE_KEY
into #StaffCourses
from ndesql04.STG_MART.SCHOLWHS.STAFF_SNAPSHOT ss with(nolock)
inner join ndesql04.STG_MART.SCHOLWHS.STUD_GRADES sg with(nolock)
        on sg.STAFF_KEY = ss.STAFF_KEY
        and sg.SCHOOL_YEAR = ss.SCHOOL_YEAR
where @School_Year = ss.SCHOOL_YEAR

select spi.NDE_Staff_ID,
        sg.COURSE_KEY
from DataCenter.dbo.vwStaff_Personal_Information spi with(nolock)
inner join #StaffCourses sc with(nolock)
        on sc.STAFF_ID = spi.NDE_Staff_ID
where spi.DataYears = @Datayears
```

8.1.4 Use for Optimizing Joins

a) Similar to the use of #temp tables in linked server queries, if you have a complex and slow set of joins in a query between multiple large tables you may be able to speed up the query by replacing some of the joins with a single flattened #temp table that presents only the columns needed for replacements in the SELECT, WHERE and JOIN clauses.

```
select ss.SCHOOL_YEAR,
        ss.STAFF_ID,
        c.COURSE_ID
into #StaffCourses
from STG_MART.SCHOLWHS.STAFF_SNAPSHOT ss with(nolock)
inner join STG_MART.SCHOLWHS.STUD_GRADES sg with(nolock)
        on sg.STAFF_KEY = ss.STAFF_KEY
        and sg.SCHOOL_YEAR = ss.SCHOOL_YEAR
inner join STG_MART.SCHOLWHS.COURSE c with(nolock)
        on c.COURSE_KEY = sg.COURSE_KEY
        and c.SCHOOL_YEAR = sg.SCHOOL_YEAR
        and c.DISTRICT_KEY = sg.DISTRICT_KEY
        and c.LOCATION_KEY = sg.LOCATION_KEY

select sds.SchoolYear,
        sds.NDE_Staff_ID,
        sc.COURSE_ID
from STG_MART_NDE.dbo.DataCollections_Staff_Demographics_Staging sds with(nolock)
inner join #StaffCourses sc with(nolock)
        on sc.SCHOOL_YEAR = sds.SchoolYear
        and sc.STAFF_ID = sds.NDE_Staff_ID
```

8.2 Inner joins vs. Outer joins

Best Practices:

All joins should explicitly state if they are INNER or LEFT OUTER joins for ease of maintenance, do not use comma-separated implicit joins. A key thing to remember is that, given which table you choose in the FROM clause, using an INNER join will mean that your result set will only ever have the same number of rows as the first table or less. Using a LEFT OUTER join, however, will result in the same number of rows as the first table (if the joins is made carefully and you use a DISTINCT) or you may end up with many more rows - possibly by a factor of the number of rows in the first table multiplied by the number of rows in the joined table.

8.2.1 NoLock Hint

a) Something that is generally recommended is to always use the With(NoLock) hint with any table in a FROM or JOIN clause. This may help prevent deadlocking errors, and helps with readability by identifying all tables used in a query at a glance. The hint can be used with #temp tables as well. Note there may be situations where you want the default locking behavior if you are working with a table that is subject to constant updating.

8.2.2 Following Indexes

a) In general, if you aren't sure about the optimal way to join your tables then you should review the columns present in the primary key or indexes on the tables, and pick a subset of columns from that list if possible.

8.2.3 IsNull() and DISTINCT for Outer Joins

a) Almost all queries that use outer joins should include either a DISTINCT hint in the SELECT clause or should have a GROUP BY clause, this will avoid unexpected duplicates caused by the row multiplication. Even with a DISTINCT hint you can still get unexpected duplicates because of the columns you happened to include in the SELECT clause, therefore it is recommended to only include the minimum number of columns needed, rather than selecting *all columns from one or more tables.

b) This behavior can be used to your advantage too. One common example - if you have a primary table which contains the exact set of rows you want in your result set, and you have a secondary table that may not have a match for every row in the first table. You can use a DISTTINCT combined with the IsNull() function to preserve the original set of rows:

```
select agy.AGENCYID,
        agy.NAME,
        isnull(ss.CAREER_EDUCATION_DESC,'No Career Ed Program') as 'CareerEd'
from dbo.AGENCIES agy with(nolock)
left outer join NDE_MART.dbo.STUDENT_SUMMARY ss with(nolock)
        on ss.DISTRICT_CODE = left(agy.AGENCYID,7)
        and ss.SCHOOL_CODE = agy.SCHOOL
        and ss.DATAYEARS = agy.DATAYEARS
and ss.CAREER_EDUCATION_CODE = 1          --Yes
```

8.2.4 Linked Servers

a) Try to avoid joining a table on one server with a table on a linked server in a complex query if possible. See 8.1.3 above.

8.2.5 Table Updating Efficiently

a) Joins may be used in UPDATE statements as well, by formatting your update in a slightly different manner using table aliases. This option, combined with CASE statements, etc., can be used to update a table in a single statement versus having to do multiple repetitive updates on the same table which cannot be automatically optimized and may be less efficient overall:

```
update de
set de.DISTRICT_KEY = case when err.STATUS = 'I'
                           then '00-0000'
                           else de.DISTRICT_KEY end
from dbo.DISTRICT_ERROR de with(nolock)
inner join dbo.ERROR err with(nolock)
        on err.ERROR_CODE = de.ERROR_CODE
```

8.3 Group By/Having clauses

Best Practices:

Using a GROUP BY clause can be used in a manner similar to a SELECT DISTINCT, but grouping is recommended only when your data has a well-defined hierarchy or when you specifically need to create sums or averages.

8.3.1 Counts

a) The Count function is commonly used, but less known in SQL Server is the ability to combine it with a DISTINCT to get a true unique count which is often what is actually desired. In the example below, the first count will actually give you a total of all

sections/classrooms being taught (essentially the logic of the count matches the primary key of the table) while the second count gives you the actual courses being offered.

```
select count(nde_student_id) as FullCount,
        count(distinct nde_student_id) as StudentCount
from StudentRecords.dbo.vw_studentGrades
```

8.3.2 Max/Min

a) The Max and Min functions have obvious uses for finding numerical minimums/maximums, but the Max function is also recommended for selecting additional columns that should be unique to the set of columns in the Group By clause but aren't logical choices to be in the Group By clause itself. In other words, it is recommended that you don't automatically put all your selected columns in the Group By clause:

```
select
        DATAYEARS,
        COUNTY,
        DISTRICT,
        NAME,
        count(distinct SCHOOL) as SchoolCount
from DataWarehouse.dbo.AGENCIES agy with(nolock)
group by DATAYEARS,COUNTY,DISTRICT,NAME
order by DATAYEARS,COUNTY,DISTRICT
```

b) Instead use the logical grouping (county/district numbers are unique to a district):

```
select
        DATAYEARS,
        COUNTY,
        DISTRICT,
        max(NAME) as NAME,
        count(distinct SCHOOL) as SchoolCount
from DataWarehouse.dbo.AGENCIES agy with(nolock)
group by DATAYEARS,COUNTY,DISTRICT
order by DATAYEARS,COUNTY,DISTRICT
```

c) This also helps to avoid unexpected duplicate results due to unforeseen data, such as when a district changes its name between years.

d) The Max function can also be used to bring together data that, due to different data sources, must come from multiple different queries. The queries can be brought together with a UNION ALL inside a subquery to get a single combined result set:

```
select subquery.STUDENT_ID,
        max(subquery.FRL) as FRL,
        max(subquery.CTEParticipant) as CTEParticipant
from (
        select ss.STUDENT_ID,
                1 as FRL,
                0 as CTEParticipant
        from NDE_MART.dbo.STUDENT_SUMMARY ss with(nolock)
        where ss.FOOD_PROGRAM_ELIGIBILITY_CODE in(1,2,3)

        union all

        select s.STUDENT_ID,
                0 as FRL,
                1 as CTEParticipant
        from STG_MART.SCHOLWHS.PROGRAMS_FACT pf with(nolock)
        inner join STG_MART.SCHOLWHS.PROGRAMS_CODE pc with(nolock)
                on pf.PROGRAMS_KEY = pc.PROGRAMS_CODE
                and pf.SCHOOL_YEAR = pc.SCHOOL_YEAR
        inner join STG_MART.SCHOLWHS.STUDENT s with(nolock)
                on s.STUDENT_KEY = pf.STUDENT_KEY
                and s.SCHOOL_YEAR = pf.SCHOOL_YEAR
        where pc.PROGRAMS_CODE like 'CE%'
) subquery
group by subquery.STUDENT_ID
```

8.3.3 Summaries

a) The Sum function is often used with groups to do standard summations, but it can also be useful when combined with a subquery that uses the same table as the main query. This can be used to get the correct answer you want for the sum while avoiding complications caused by duplicate rows that could arise from the grouping in the main query. For example:

```
select vcc.Datayears,
        vcc.course_code,
        CourseCounts.StudentCount / count(distinct vcc.Agencyid) as AgencyAverage
from dbo.vw_Curriculum_Combined vcc with(nolock)
inner join (
        select datayears,
                Agencyid,
                course_code,
                sum(Total_Students) as StudentCount
        from dbo.vw_Curriculum_Combined with(nolock)
        group by datayears, Agencyid, course_code
) CourseCounts
        on vcc.Datayears = CourseCounts.Datayears
        and vcc.Agencyid = CourseCounts.Agencyid
        and vcc.course_code = CourseCounts.course_code
group by vcc.datayears, vcc.course_code
```

8.3.3.1 Using CASE for conditional sums or counts

a) The Sum function can be combined with Case statements in order to do multiple or complex counts in a single query.

```
select s.DATAYEARS,
        sum(case when s.EXPERIENCE_TOTAL < 2 then 1 else 0 end) as Year1Count,
        sum(case when s.EXPERIENCE_TOTAL = 2 then 1 else 0 end) as Year2Count,
        sum(case when s.EXPERIENCE_TOTAL = 3 then 1 else 0 end) as Year3Count,
        sum(case when s.EXPERIENCE_TOTAL = 4 then 1 else 0 end) as Year4Count,
        sum(case when s.EXPERIENCE_TOTAL = 5 then 1 else 0 end) as Year5Count
from dbo.STAFF_20092010 s with(nolock)
group by s.DATAYEARS
```

b) Be aware that since this example does not use a distinct count that it will only work when you know your query already has distinct rows.

8.3.4 HAVING clause

a) The HAVING clause can is used with a GROUP BY and works similarly to a standard WHERE clause, except that you can perform logic on data after it has been used in an aggregate function (Sum, Count, etc.). For example:

```
select [CoDist] ,[FiscalYrnbr] ,[SchoolNbr] ,nssrs ,COUNT(rowID)
from [CNPNET].[dbo].[DirectCertificationDetail]
where [FiscalYrnbr] = 2015
group by [CoDist]
    ,[FiscalYrnbr]
    ,[SchoolNbr]
    ,nssrs
 having COUNT(rowID) >1
```

8.4 Subqueries (SELECT clause)

Best Practices:

A subquery is any fully defined SELECT statement containing within parentheses in another query. They can appear in multiple places as noted below. However, it is recommended to avoid using them in the SELECT clause itself as these queries often have poor performance due to them being executed over and over for the maximum possible number of rows returned. Below is an example to **avoid**:

```
select SCHOOL_YEAR,
        DISTRICT_KEY,
        ERROR_CODE,
        (select status
                from dbo.ERROR e with(nolock)
                where e.ERROR_CODE = de.ERROR_CODE
        ) as ErrorStatus
from dbo.DISTRICT_ERROR de with(nolock)
```

8.4.1 Subqueries with CASE statements

a) Using subqueries within case statements in the SELECT clause seems like a natural idea (see example above in 8.4). However, these queries can often be rewritten by changing the necessary query into a join:

```
select de.SCHOOL_YEAR,
      de.DISTRICT_KEY,
      de.ERROR_CODE,
      e.status as ErrorStatus
from dbo.DISTRICT_ERROR de with(nolock)
inner join dbo.ERROR e with(nolock)
on e.ERROR_CODE = de.ERROR_CODE
```

b) If that can't be done, then the data should be stored in a #temp table ahead of time as well to increase efficiency:

```
select ERROR_CODE,status
into #ERROR
from dbo.ERROR e with(nolock)

select SCHOOL_YEAR,
      DISTRICT_KEY,
      ERROR_CODE,
      (select status
            from #ERROR e with(nolock)
            where e.ERROR_CODE = de.ERROR_CODE
      ) as ErrorStatus
from dbo.DISTRICT_ERROR de with(nolock)
```

8.5 Subqueries (WHERE clause) / EXISTS clause

Best Practices:

a) A subquery may also be used within a WHERE clause in a SELECT statement:

```
select de.SCHOOL_YEAR,
       de.DISTRICT_KEY,
       de.ERROR_CODE
from dbo.DISTRICT_ERROR de with(nolock)
where (select status
       from dbo.ERROR e
       where e.ERROR_CODE = de.ERROR_CODE) = 'A'
```

b) But these subqueries can usually be converted into an EXISTS clause, which is typically more efficient on the server and can make a noticeable difference in the runtime for a complex query:

```
select de.SCHOOL_YEAR,
       de.DISTRICT_KEY,
       de.ERROR_CODE
from dbo.DISTRICT_ERROR de with(nolock)
where exists( select *
       from dbo.ERROR e
       where e.ERROR_CODE = de.ERROR_CODE and status = 'A')
```

8.5.1 In vs. Exists

a) Similar to the idea of replacing a subquery with an EXISTS clause above, an IN clause:

```
select de.SCHOOL_YEAR,
       de.DISTRICT_KEY,
       de.ERROR_CODE
from dbo.DISTRICT_ERROR de with(nolock)
where de.ERROR_CODE in( select ERROR_CODE
       from dbo.ERROR e
       where status = 'A')
```

b) Should most likely be converted to an EXISTS clause where possible even though the code may not be as intuitive.

8.5.2 Deleting

a) One place where this type of subquery is useful is in DELETE statements, since joins cannot be used. Below is an example of a deletion that uses a subquery to do complex logic that would replace a JOIN in a typical query:

```
delete from dbo.DISTRICT_ERROR
where ERROR_CODE in( select ERROR_CODE
                     from dbo.ERROR e
                     where status = 'A')
```

8.6 Subqueries (nested join)

Best Practices:

a) The place most recommended to use subqueries is within the joins in a SELECT statement. They are indispensable for creating complex pieces of logic that can be used with other tools to accomplish things in one query that might take dozens otherwise. Even though that single query may take some time to run, it is almost always faster than the alternative queries. For example this single query:

```
select    gc.CareerField, gc.Cluster, c.GradeLevel, c.DataYears,
          count(distinct left(c.Agencyid,7)) as NumberOfDistricts,
          count(distinct c.agencyid) as NumberOfSchools
from ndesql04.pos.dbo.POS_GuideCourses gc
inner join (        select agencyID,course_code,'09' as GradeLevel,DataYears
                    from dbo.vw_Curriculum_Combined
                    where G09 = 'Y'
                    union all
                    select agencyID,course_code,'10' as GradeLevel,DataYears
                    from dbo.vw_Curriculum_Combined
                    where G10 = 'Y'
                    union all
                    select agencyID,course_code,'11' as GradeLevel,DataYears
                    from dbo.vw_Curriculum_Combined
                    where G11 = 'Y'
                    union all
                    select agencyID,course_code,'12' as GradeLevel,DataYears
                    from dbo.vw_Curriculum_Combined
                    where G12 = 'Y') c
          on gc.CourseCode = c.course_code
inner join DataWarehouse.dbo.AGENCIES agy
          on c.Agencyid = agy.AGENCYID
          and c.DATAYEARS = agy.DATAYEARS
          and agy.DISTRICT_TYPE_CODE in ('p','so')
group by gc.CareerField, gc.Cluster, c.GradeLevel, c.DataYears
```

b) Also see examples in 8.3.3

8.6.1 Grouped summaries

a) Joined subqueries can be used in combination with grouping and the min/max/sum/count functions to create complex queries. See examples above in 8.3.3

9: Environments – (To be developed, Reference Environments Document)

10: Database Descriptions and Purposes

Standards:

a) Database Definitions and Purposes are stored in the NDESQL01.DataWarehouse under 'DBO.DATABASE_DESCRIPTIONS'

11: Accounts and Passwords

11.1 Three types of accounts:

Standards:

a. System Accounts: SQL Server login accounts used by computer systems and reports to access databases
b. User Accounts: Windows Login accounts used by employees and contractors to accomplish database tasks.
c. Link Server Accounts: It links a Windows account to a SQL Server account when it crosses servers to facilitate database work between two or more servers

11.2 Naming of Accounts:

Standards:

a. System Accounts:

    [ProjectName]Read – read only / execute - reports
    [ProjectName]Write – Read/ write / execute - Systems

b. User Accounts:

    Established by Windows / Active Directory

c. Link Server Account:

    {WindowsAccount}_LS

11.3 Passwords:

Standards:

a. Passwords must comply with the NITC 8-301 Password Standard Policy.

11.4 Security:

Standards:

a. Each system or user account will be warranted security access and will have to be requested through the Database Access Form.

12: Jobs

12.1 Job Naming:

Standards:

a. Job name consists of the job objective
b. Specific details of the job are not to be listed in the name.
c. Omit timeframes in job naming (Daily / Nightly)
d. Name it noun verb to keep like jobs together.

   EXAMPLE: DCVMS Match Load

e. Jobs contained within a series of job

   Must contain the series number at the end – PART 1, PART 2
   If job step is added between, add a letter – PART 1B

f. When creating a series from an original job, do not rename the initial job to include PART 1. Just continue the series and note in description field of the job the link between the jobs.

12.2 Job Description Field:

Standards:

a. Add who changed or created the job
b. Specific details of the job are listed in the Description field of the job.
c. Description of what job purpose
d. Dependencies to or from other jobs
e. Date and time stamp of job changes
f. Newest details added to the bottom

   Example:

   Author: Travis Rhoden
   Creation date: April 2, 2013
   Definition / Purpose:
   Change history:
   April 10, 2013 - Travis Rhoden – Added Procedure Load_HAL_Data at step 4
   August 11, 2013 – Changed the data year to 20122013

13: Extract, Transform, and Load (ETL) - **(To be developed – Reference ETL Document)**

14: Documentation - **(To be developed – Reference Documentation Document)**

Provided are suggestions but not inclusive:

- Data Sources – Internal or External
- Data Flow
- Procedure Order
- Dates of completion or Schedules

Glossary

Best Practices – These are suggestions based on previous situations that worked best for our organization.

Standards – These are items that must be followed.

Lookup Table - Auxiliary table that holds static data, indicate with a 'lu' prefix in DataWarehouse

Intersector Table – Table used to break down Many to Many relationships into 1 to 1.

Primary Key – Unique Value(s) for each record, could be composite key or identity field.

Foreign Key - A field (or collection of fields) in one table (Child Table) that uniquely identifies a row of another table (Parent Table)

Parent Table - A table that contains a primary key that is referenced by at least one foreign key in another table (Child Table).

Child Table - A table that contains a foreign key that references the primary key in the Parent Table.

System Accounts - SQL Server login accounts used by computer systems and reports to access databases

User Accounts - Windows Login accounts used by employees and contractors to accomplish database tasks.

Link Server Accounts - Links a Windows account to a SQL Server account when it crosses servers to facilitate database work between two or more servers

Reserved or Forbidden Words Appendix

You cannot use these words as column or field names, because they are used within MS SQL
These are the reserved words for MS SQL:

ADD	CURRENT_TIMESTAMP	GROUP	OPENQUERY	SERIALIZABLE
ALL	CURRENT_USER	HAVING	OPENROWSET	SESSION_USER
ALTER	CURSOR	HOLDLOCK	OPTION	SET
AND	DATABASE	IDENTITY	OR	SETUSER
ANY	DBCC	IDENTITYCOL	ORDER	SHUTDOWN
AS	DEALLOCATE	IDENTITY_INSERT	OUTER	SOME
ASC	DECLARE	IF	OVER	STATISTICS
AUTHORIZATION	DEFAULT	IN	PERCENT	SUM
AVG	DELETE	INDEX	PERM	SYSTEM_USER
BACKUP	DENY	INNER	PERMANENT	TABLE
BEGIN	DESC	INSERT	PIPE	TAPE
BETWEEN	DISK	INTERSECT	PLAN	TEMP
BREAK	DISTINCT	INTO	PRECISION	TEMPORARY
BROWSE	DISTRIBUTED	IS	PREPARE	TEXTSIZE
BULK	DOUBLE	ISOLATION	PRIMARY	THEN
BY	DROP	JOIN	PRINT	TO
CASCADE	DUMMY	KEY	PRIVILEGES	TOP
CASE	DUMP	KILL	PROC	TRAN
CHECK	ELSE	LEFT	PROCEDURE	TRANSACTION
CHECKPOINT	END	LEVEL	PROCESSEXIT	TRIGGER
CLOSE	ERRLVL	LIKE	PUBLIC	TRUNCATE
CLUSTERED	ERROREXIT	LINENO	RAISERROR	TSEQUAL
COALESCE	ESCAPE	LOAD	READ	UNCOMMITTED
COLUMN	EXCEPT	MAX	READTEXT	UNION
COMMIT	EXEC	MIN	RECONFIGURE	UNIQUE
COMMITTED	EXECUTE	MIRROREXIT	REFERENCES	UPDATE
COMPUTE	EXISTS	NATIONAL	REPEATABLE	UPDATETEXT
CONFIRM	EXIT	NOCHECK	REPLICATION	USE
CONSTRAINT	FETCH	NONCLUSTERED	RESTORE	USER
CONTAINS	FILE	NOT	RESTRICT	VALUES
CONTAINSTABLE	FILLFACTOR	NULL	RETURN	VARYING
CONTINUE	FLOPPY	NULLIF	REVOKE	VIEW
CONTROLROW	FOR	OF	RIGHT	WAITFOR
CONVERT	FOREIGN	OFF	ROLLBACK	WHEN
COUNT	FREETEXT	OFFSETS	ROWCOUNT	WHERE
CREATE	FREETEXTTABLE	ON	ROWGUIDCOL	WHILE
CROSS	FROM	ONCE	RULE	WITH
CURRENT	FULL	ONLY	SAVE	WORK
CURRENT_DATE	GOTO	OPEN	SCHEMA	WRITETEXT

CURRENT_TIME GRANT OPENDATASOURCE SELECT

These are the reserved words in MS SQL Field names:

binary	decimal	nchar	smalldatetime	tinyint
bit	float	ntext	smallint	uniqueidentifier
char	image	numeric	smallmoney	varbinary
cursor	int	nvarchar	text	varchar
datetime	money	real	timestamp	